The gnufi User and Developer Documentation

For version 0.0, 24 October 2006

Johan Rydberg <jrydberg@gnu.org>

Table of Contents

1 Introduction

GNUFI is a free implementation of the UEFI specification. UEFI defines a set of services and an environment that can be used to boot into an operating system.

The design goals for GNUFI is that is should be (1) portable, (2) extendable and (3) specification confirming. To achieve those goals the following concepts are introduced:

Loaders
: At configuration time different firmware loaders can be specified (See Chapter 3 [Loaders], page 3.). This enables the firmware to be run in different pre-existing environments. An example of a firmware loader could be '`i386-legacy`'; a loader that loads the firmware in a legacy BIOS environment. Another example loader could be '`ppc-uboot`'; a loader that loads the firmware using the U-Boot bootloader on a PPC architecture.

Kernel scripts
: Per-board configuration scripts, called kernel scripts or simply kscripts, are used to enable the user to select what services to include in a firmware image, and how to connect those services to devices. The kscripts are also used to setup fall-back console I/O in case that the boot manager can not initialize the preferred console.

Modules
: Most services are implemented as modules, this enables the user of the firmware to pick with great detail what services he or she requires for their environment.

2 Building GNUFI

2.1 Building The Firmware Kernel

The GNUFI build system uses a `configure` script generated by GNU Autoconf.

Before running the configure script, a firmware kernel loader has to be picked. See Chapter 3 [Loaders], page 3, for a complete list of available loaders. The loader is specified using the '`--with-loader`' option to the configure script.

Suggestion: Build the firmware in a separate directory from the sources.

```
gnufi-0.0$ mkdir build && cd build
gnufi-0.0/build$ ../configure --with-loader=i386-multiboot
...
gnufi-0.0/build$ make
...
gnufi-0.0/build$
```

The above example configures and builds a firmware kernel suitable to be loaded by a multiboot compliant boot loader, such as GNU GRUB. It also builds a set of runtime modules and a few utilities.

2.2 Building A Firmware Image

The step after configuring and building the firmware kernel is to build a firmware image suitable for the system it should be run on. This is done using the '`gnufi-mkimage`' utility that was built alongside the kernel.

The example below uses the '`gnufi-mkimage`' utility to create a firmware image that is suitable to be used on Intel 440BX based systems:

```
$ gnufi-mkimage -o firmware.image i386-440bx.kscript
```

How the image is installed and started is loader specific. See Chapter 3 [Loaders], page 3.

2.3 The 'gnufi-mkimage' Utility

The '`gnufi-mkimage`' utility takes a kernel script file as argument (See Chapter 4 [Kernel scripts], page 4.) and creates a firmware image based on its contents. The utility also takes the following options;

'`-o` *`FILE`*' Specify output file.

'`-d` *`DIR`*' Specify where modules specified in the configuration file can be found.

2.4 Available Kernel Scripts

Below is a list of board specific kernel scripts distributed with GNUFI:

'`i386-440bx.kscript`'
: Intel 440BX based boards.

3 Firmware Loaders

GNUFI is designed so that the firmware can be loaded in several different ways, suiting the desired environment. A '`loader`' is responsible for loading the firmware kernel into memory and collecting information about the environment before passing control to the firmware kernel.

3.1 i386 Multiboot loader

TODO

4 Board Specific Configuration Scripts

Board specific configuration scripts, called kernel scripts or simply kscripts, are used to enable the user to select what services to include, and how to connect those services to devices. The scripts contain a sequence of operations to perform while starting the firmware. An operation is specified by a single operation command word and a set of arguments. All lines that start with a '`#`' are treated as comments. The following operations are available:

`load` *MODULE*
: Load and start module specified by *MODULE*. The module can either be a driver or an application.

`connect` *PATH*
: Connect device path specified by *PATH* to a (or several) suitable drivers picked by the firmware. The last component of the path will be recursively connected, meaning that any children that it may create will also be connected.

`set` *VARIABLE PATH*
: Set variable *VARIABLE* to the handle specified by *PATH*. The following variables are available:

  `stdout`
  `stderr` Sets handles for standard output and standard error output.

  `stdin` Set handle for standard input.

Below is a complete (bogus) example of a kscript that loads a set of modules, connects the PCI IDE controller, which will also connect any possible disks that it detects. After that it will set console handles to the first serial console on the PCI ISA bridge.

```
# board kscript: bogus-example.kscript
load cpuio-i386.efi
load pci-rb+i386t1.efi
load pci-bus.efi
load pci-ide.efi
load disk-io.efi
load pci-isa.efi
load ser-ns16550.efi
# connect IDE controller
connect /piciroot(0)/pci(1,1)
# connect stdout and stderr
set stdout /piciroot(0)/pci(0,1)/acpi(PNP0503,0)
set stderr /piciroot(0)/pci(0,1)/acpi(PNP0503,0)
# connect stdin
set stdin /piciroot(0)/pci(0,1)/acpi(PNP0503,0)
```

5 Libraries

5.1 libefi: Standard UEFI API

The '`libefi`' library contains the functions for the standard UEFI interface alongside a few convenience functions. Include the file `<gnu/efi/api.h>` to get prototypes for the functions.

5.1.1 Protocol Interface Related Functions

TODO: overall description

`efi_status_t efi_install_protocol_interface` (*efi_handle_t* ***handle**, *efi_guid_t* ***guid**, *efi_interface_type_t* **type**, *void* ***interface**) [Function]

Attach a protocol interface to handle **handle*. The protocol is identified by *guid*. If **handle* is `NULL`, a new handle will be allocated and returned.

`efi_status_t efi_uninstall_protocol_interface` (*efi_handle_t* **handle**, *efi_guid_t* ***guid**, *void* ***interface**) [Function]

Uninstall a protocol instance. If this was the last protocol associated with the handle, the handle will be destroyed.

5.1.2 Device Path Related Functions

These functions are user to manage device paths in different ways. Most of them are not part of the UEFI interface, but implemented as convenience functions in the library.

`efi_status_t efi_device_path_add_to_handle` (*efi_handle_t* **handle**, *efi_device_path_t* ***path**) [Function]

Add device path specified by *path* to the device handle *HANDLE*.

`efi_device_path_t* efi_device_path_get_from_handle` (*efi_handle_t* **handle**) [Function]

Return device path for handle specified by *handle*, or `NULL` in case of error.

`efi_device_path_t* efi_device_path_build` (*efi_boolean_t* **free_them**, ...) [Function]

Build a device path from a given list of path nodes. The list is NULL terminated. If *free_them* is true, release memory for all given nodes.

`efi_device_path_t* efi_device_path_iterate` (*efi_device_path_t* ***pathp*) [Function]

Iterate nodes of the device path. **pathp* should be set to point to the path that is to be iterated. `NULL` will be returned when the end of the path has been reached.

5.1.3 String Functions

`efi_int_t efi_wstrcmp` (*efi_char16_t* ***s1**, *efi_char16_t* ***s2**) [Function]

Compare unicode strings *s1* and *s2* and return the difference.

`efi_int_t efi_wstrncmp` (*efi_char16_t* ***s1**, *efi_char16_t* ***s2**, *efi_uint_t* **n**) [Function]

Compare unicode strings *s1* and *S2*, and return the difference. Only compare *n* number of characters.

`efi_int_t efi_wstricmp` (*efi_char16_t *s1*, *efi_char16_t *s2*) [Function]
Compare strings and return the difference, independent of case.

`efi_char16_t* efi_wstrcpy` (*efi_char16_t *d*, *efi_char16_t *s*) [Function]
Copy string specified by *s* to the memory buffer pointed to by *d*. Return *d*.

`efi_char16_t* efi_wstrncpy` (*efi_char16_t *dst*, *const efi_char16_t *src*, *efi_uint_t size*) [Function]
Copy *size* characters from string *src* to string buffer *dst*. Return *dst*

`efi_uint_t efi_wstrlen` (*efi_char16_t *s*) [Function]
Return length of given string *s*.

5.1.4 Parsing of Options

The 'efi' library contains functionality for parsing options. The parsing framework is modeled after `argp` from the GNU C library. The application defines a list of valid options, and pass them together with a parse hook to the `efi_argp_parse` method. The hook will get invoked during the parsing with information about what option was just parsed. Definitions and prototypes is available in 'gnu/efi/util/argp.h'.

The options is defined as a NULL terminated array of `efi_argp_option_t` structures. The `efi_argp_option_t` structure has the following members:

`efi_char16_t* longarg` [argp Option]
Long argument name of the option, if any. Long arguments are argumnets that start with '--'. For example '--output'.

`int shortarg` [argp Option]
Short argument name of the option, if any. Short argument are arguments that start with '-'. For example '-o'.

`int flags` [argp Option]
Option flags. The only flag currently available is `EFI_ARGP_ARG_OPTIONAL` which defines that the option takes an optional argument.

`int key` [argp Option]
A key that can be used by the parser hook to identify the option.

`efi_char16_t* doc` [argp Option]
A documentation string that describes the option. Mandatory.

`efi_char16_t* arg` [argp Option]
Name of the argument that the option takes. Should be in uppercase.

FIXME: If neither *longarg* nor *shortarg* is specified in the option is treated as an mandatory argument to the program. If no option is defined for the program argument, the hook will be invoked with *opt* as NULL but a valid *arg* argument.

FIXME: When all options has been parsed the parser will invoke the hook with *opt* and *arg* set to NULL to signal that all options has been parsed.

The signature of the hook:

`efi_status_t efi_argp_parse_hook_t` (*void* `*data`, *efi_argp_option_t* `*opt`, *efi_char16_t* `*arg`) [Function]

Parse hook invoked from the option parser. *DATA* is a user defined pointer that was provided to *efi_argp_parse*. *opt* is the option that was pased, and *arg* is an optional argument to the option. The hook should return `EFI_SUCCESS` if it could handle the option.

Caution: The option parser uses the first argument passed to `efi_argp_parse` as the program name.

`efi_status_t efi_argp_parse` (*int* `argc`, *efi_char16_t* `**argv`, *efi_argp_option_t* `*options`, *efi_argp_parse_hook_t* `hook`, *void* `*data`) [Function]

Parse options and invoke hook *hook* with the parsed option. *options* is a list of options that should be parsed. *argc* is the number of arguments passed to the program. *argv* is an array of all arguments. *data* is a user defined pointer that will be passed to the hook.

5.1.4.1 Example usage of efi_argp_parse

The small example below shows a program that accepts two options: '`--count`' and '`--verbose`', with short versions '`-c`' and '`-v`' respectively.

```
static efi_argp_option_t options = []
  {
    { L"count", 'c', 0, 0, L"Forward direction", L"NUM" },
    { L"verbose", 'v', 0, 0, L"Verbose output", NULL },
    { 0, 0, 0, 't', L"Text to display", L"TEXT" },
    { 0, 0, 0, 0, 0, 0 }
  };

int count = 10;
int verbose = 0;
efi_char16_t *text = NULL;

static efi_status_t
parse (void *data, efi_argp_option_t *opt, efi_char16_t *arg)
{
    switch (opt->shortarg)
      {
      case 'c':
        count = efi_wstrtoul (arg, NULL, 0);
        break;
      case 'v':
        verbose = 1;
        break;
      }
    switch (opt->key)
      {
      case 't':
        text = arg;
        break;
      }
  return EFI_SUCCESS;
}

efi_status_t
```

```
efi_main (efi_handle_t image, int argc, efi_char16_t **argv)
{
  efi_status_t err;
  int i;

  err = efi_argp_parse (argc, argv, options, parse, 0);
  if (err)
    return err;

  for (i = 0; i < count; i++)
    if (verbose)
      efi_printf (L"%s\n", text);

  return EFI_SUCCESS;
}
```

5.1.5 Memory Management

Besides the functions that the UEFI makes available for memory management, the 'efi' library provides a set of convenience functions to make it easier to allocate and free memory.

To allocate pooled memory the `efi_malloc` function can be used. The difference from `efi_allocate_pool` is that `efi_malloc` returns a pointer to the allocated memory. Memory allocated with `efi_malloc` is like any other pooled memory freed with `efi_free_pool`.

`void* efi_malloc` (*efi_memory_type_t type*, *efi_uint_t size*) [Function]
: Allocate *size* bytes of pooled memory from memory pool specified by *type*. Return pointer to allocated memory, or `NULL` if memory could not be allocated.

The `efi_palloc` and `efi_pfree` functions operate on *page memory*, meaning that the handle non-pooled memory. `efi_palloc` allocates memory and sets it to a specified memory type, using the given allocation scheme. `efi_pfree` releases memory allocated with either `efi_palloc` or `efi_allocate_pages`.

`void* efi_palloc` (*efi_allocate_type_t allocate_type*, *efi_memory_type_t mt*, *efi_uint_t size*) [Function]
: Allocate memory using allocation scheme *allocate_type* and memory type *mt*. Allocate *size* bytes of memory.

`void efi_pfree` (*void *p*, *efi_uint_t size*) [Function]
: Free pages pointed to by *p*. *size* is the number of bytes allocated.

5.2 libefi-fshelp: File System Help Library

The 'libefi-fshelp' library can be used by drivers that want to implement the 'simple file system' protocol. The library presents a complete interface, but communicates with the file system backend through a minimal API. The interface is defined in the 'gnu/efi/util/fshelp.h' header.

The file system backend defines a type 'efi_fshelp_node_t' that represents a node in the file system. The type is fully opaque to the library. Nodes are reference counted. The library can at any time obtain an extra reference by calling the `reference` operation on the node, or release a reference by invoking `release`. It is up to the file system implementation to maintain references.

`efi_status_t efi_fshelp_install_protocol_interface` [Function]
(*efi_handle_t* `controller`, *efi_fshelp_t* *`fshelp`, *struct efi_fshelp_ops* *`ops`)
Install all the protocol interfaces that the library defines on the controller specified by *controller*. *fshelp* is the fshelp instance that will be used through out the lifetime of the protocols. *ops* is the file system backend operations vector.

The `efi_fshelp_ops` vector contains the following operations:

`efi_status_t mount` (*efi_fshelp_t* *`fshelp`, *efi_fshelp_node_t* **`root`) [fshelp Operation]
Try to mount file system and return a reference to node in **root*.

`void reference` (*efi_fshelp_node_t* *`node`) [fshelp Operation]
Obtain reference to node specified by *node*.

`void release` (*efi_fshelp_node_t* *`node`) [fshelp Operation]
Release reference to node specified by *node*, possible destroy it.

`efi_status_t getinfo` (*efi_fshelp_node_t* *`node`, *efi_file_info_t* *`info`) [fshelp Operation]
Retreive information about the node *node* and store it in the buffer specified by *info*. The backend may in turn cache the information for higher performance.

`efi_status_t setinfo` (*efi_fshelp_node_t* *`node`, *efi_file_info_t* *`info`) [fshelp Operation]
Set information about the node *node* given the information in *info*.

`efi_status_t readdir` (*efi_fshelp_node_t* *`dir`, *efi_uint64_t* *`fpos`, *efi_fshelp_node_t* **`nodep`, *efi_char16_t* **`namep`) [fshelp Operation]
Read a directory entry at the position specified by **fpos* from node *dir*. If no more directory entries could be read, the operation should return `EFI_NOT_FOUND`. A reference to the node in the read directory entry is returned in **nodep*, and the name of the entry in **namep*. The caller is resposible for releasing the reference to **nodep* and freeing the memory of **namep*. The position **fpos* should be updated on a successful read.

`efi_status_t read` (*efi_fshelp_node_t* *`node`, *efi_uint64_t* `fpos`, *efi_uint_t* *`size`, *void* *`buffer`) [fshelp Operation]
Read data from the node *node* at position given by *fpos*. The amount of data to read is determined by **size*, which will be updated with the actual amount of data read on return. On return, if **size* is zero there is no more data to read. The data will be read into buffer *buffer*.

The library also implements a set of helper functions that can be used to more easily implement some of the fshelp operations.

`efi_status_t efi_fshelp_read_helper` (*efi_fshelp_node_t* *`node`, *efi_fshelp_read_hook_t* `read`, *void* *`read_data`, *efi_uint64_t* `pos`, *efi_uint_t* *`size`, *void* *`buffer`, *efi_fshelp_block_hook_t* `block`, *efi_uint64_t* `filesize`, *int* `log2blocksize`) [Function]
Read **size* bytes from the node *node* into the buffer *buffer*, beginning with the block *pos*. *read* hook is used to read disk blocks. *read_data* will be passed to the hook as

the first argument. *block* hook is used to translate file blocks to disk blocks. The file is *FILESIZE* bytes big and the blocks have a size of *log2blocksize* (in log2, units of 512 bytes blocks).

6 GNUFI Internals

This chapter describes the internal workings of the GNUFI kernel.

6.1 Memory Management

The GNUFI kernel presents a slim memory interface that is used both by the loader and by the internals of the kernel itself.

The loader is responsible to feed memory information to the kernel. It should do this by first allocating a region of the memory to hold memory descriptor data. This is done using the `efi_mm_init` function. Characteristics and attributes for different memory regions are passed to the kernel using the `efi_mm_set` function.

Note: Characteristics for the memory descriptor region must be set manually by the loader. The kernel does *not* do that itself. The memory type should be `EFI_MEMORY_TYPE_RUNTIME_SERVICES_DATA`. The loader should also set the memory type to `EFI_MEMORY_TYPE_RUNTIME_SERVICES_CODE` and `EFI_MEMORY_TYPE_RUNTIME_SERVICES_DATA` for the kernels code and data, respectively.

Note: Memory region starting at physical address zero should not be passed to the kernel as conventional memory, nor be used as memory for the memory descriptors.

`void efi_mm_init` (*efi_physical_address_t* `addr`, *efi_uint_t* `size`) [Function]
: Initialize memory system. *addr* points to the region that will be used to manage the memory descriptors. *size* gives the size of the memory region. Note that the memory region given to this function must be also be reserved using the `efi_mm_set` function.

`efi_status_t efi_mm_set` (*efi_physical_address_t* `start`, *efi_physical_address_t* `end`, *efi_memory_type_t* `memory_type`, *efi_uint64_t* `attributes`) [Function]
: Set memory type of specified range to memory type given in *memory_type*. The region is specified by *start* and *end*.

The `efi_mm_alloc` function can be used to locate a region of memory that is suitable for allocation, given the specified allocation type. The caller is self responsible for changing memory type for the resulting region.

`efi_status_t efi_mm_alloc` (*efi_allocate_type_t* `allocate_type`, *efi_uint_t* `num_pages`, *efi_physical_address_t* `*memory`) [Function]
: Locate a suitable memory range for a memory allocation specified by *allocate_type*. *num_pages* gives the size of the requested memory.

The UEFI specification mandates the changes to the memory descriptor database must be recorded, so that it may be noted if the database has been altered between two points. This is done using a "generation counter" that is increased when an alteration is done to the database.

`efi_uint_t efi_mm_key` [Variable]
: Memory key. Updated everytime the memory descriptor database is altered.

6.2 Kernel Scripts

The kernel script is inlined in binary representation together with the GNUFI kernel in the firmware image. It is up to the '`gnufi-mkimage`' utility to merge modules and build a kscript structure that the loader can later pass to the kernel.

In binary representation is a kernel script a list of TLV-records, in native endianess. Signature of the kscript generic header;

```
struct efi_kscript
{
  efi_uint16_t type;
  efi_uint16_t subtype;
  efi_uint32_t offset;
};
```

type specifies what kind of script record it is. The last entry must be of type `EFI_KSCRIPT_END_TYPE`. *subtype* is a type specific. *offset* gives the number of bytes from this script record to the next.

`EFI_KSCRIPT_END_TYPE`
: Defines the end of the script.

`EFI_KSCRIPT_LOAD_TYPE`
: Load and start a module. The modules contents is inlined in the record. Also provided is the command line, which will be passed to the module as a load option. Signature of the record:

```
struct efi_kscript_load
{
  efi_kscript_t header;
  efi_uint32_t cmdline;
  efi_uint32_t size;
  efi_uint8_t data[0];
};
```

cmdline is an offset from the record to a NUL-terminated unicode string. *size* is the length of the module data in bytes.

`EFI_KSCRIPT_CONNECT_TYPE`
: Connect drivers to the device path that is inlined in the record.

```
struct efi_kscript_connect
{
  efi_kscript_t header;
  efi_device_path_t path;
};
```

`EFI_KSCRIPT_SET_TYPE`
: Set variable to point to the handle specified by the inlined device path.

```
struct efi_kscript_set
{
  efi_kscript_t header;
  efi_device_path_t path;
};
```

The *subtype* if the generic header specifies what variable is to be set. The following variables are available;

- EFI_KSCRIPT_SET_CONIN_SUBTYPE

- EFI_KSCRIPT_SET_CONOUT_SUBTYPE
- EFI_KSCRIPT_SET_CONERR_SUBTYPE

6.3 Internal Protocols

GNUFI uses a set of internally defined protocols to communicate between different modules. This section tries to describe them more in detail.

6.3.1 Variable Store

The internal '`variable store`' protocol is used to provide storage for variables that can be accessed through the runtime services. The protocol exposes characteristics of the store along with a set of functions used to read, clear and update the contents of the store. The store is divided into a number of banks, where each bank can be read and updated individually.

6.3.1.1 Interface

The interface provided by the '`variable store`' protocol has the following functions and variables:

`efi_uint32_t revision` [Variable Store]
: Revision of the interface, should be `EFI_VARIABLE_STORE_PROTOCOL_REVISION`.

`efi_uint32_t attributes` [Variable Store]
: Attributes of the store. See related defintions below for a list of available attributes.

`efi_uint_t bank_size` [Variable Store]
: Size in bytes of each bank in the store.

`efi_uint_t num_banks` [Variable Store]
: Number of banks that the store presents.

`efi_status_t read` (*efi_variable_store_t* ***this***, *efi_uint_t* ***bank***, *void* ***buffer***) [Variable Store]
: Read the content of bank *bank* into buffer *buffer*. The buffer must at least be able to hold the amount of data that the store defines for a bank.

`efi_status_t update` (*efi_variable_store_t* ***this***, *efi_uint_t* ***bank***, *void* ***buffer***) [Variable Store]
: Update the content of bank *bank* with the data in buffer *buffer*. The buffer msut at least hold the amount of data that the store defined for a bank.

`efi_status_t clear` (*efi_variable_store_t* ***this***, *efi_uint_t* ***bank***) [Variable Store]
: Reset content of bank *bank*.

6.3.1.2 Related Definitions

```
#define EFI_VARIABLE_STORE_GUID (efi_guid_t)  \
  { 0xdfc23a79, 0xd3f9, 0x4a0e,    \
    { 0xac, 0xcb, 0x7f, 0x11, 0xf1, 0x53, 0xf2, 0xa7 }  \
  }
```

The protocol interface contains an *attributes* variable that defines under what situations the store can be used;

`EFI_VARIABLE_STORE_NONVOLATILE`
: The store provides nonvolatile storage. Variables will survive a power-cycle.

`EFI_VARIABLE_STORE_BOOT_SERVICE`
: The store is only available while the boot services are still running.

`EFI_VARIABLE_STORE_RUNTIME`
: The store is available in runtime mode; i.e, after boot services has been terminated.

The protocol interface contains a revision member that defines the version of the interface. All future versions of the interface will be compatible. If non-compatible changes have to be introduced, a new guid will be allocated for the protocol.

`EFI_VARIABLE_STORE_PROTOCOL_REVISION`
: Protocol interface revision.

7 Concept Index

E

G

K

8 Function Index